(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) Publication number: 0 466 339 A2

(12) EUROPEAN PATENT APPLICATION

(21) Application number: 91305396.3

(51) Int. Cl.5: G06F 9/46

(22) Date of filing: 14.06.91

(30) Priority: 13.07.90 US 553203

(43) Date of publication of application:
15.01.92 Bulletin 92/03

(84) Designated Contracting States:
DE FR GB

(71) Applicant: International Business Machines Corporation
Old Orchard Road
Armonk, N.Y. 10504(US)

(72) Inventor: Disbrow, John Randolph
16500 So. Kennedy Road
Los Gatos, CA 95032(US)

(74) Representative: Moss, Robert Douglas
IBM United Kingdom Limited Intellectual Property Department Hursley Park
Winchester Hampshire SO21 2JN(GB)

(54) A method of passing task messages in a data processing system.

(57) Processors 5, 7 communicatively attached to a storage sub-system 3 place task messages for the subsystem on a queue. The processors do not have to wait on a queue lock set by another processor or sub-system whilst dequeuing a message. This is achieved by use of a double ended linked list or queue of messages having an isolation/reference point wherein an enqueuing end of the list is lockable and accessible independently from the dequeuing end of the list. The locking primitive may be of the multi-processor lock synchronizing atomic type such as TEST AND SET.

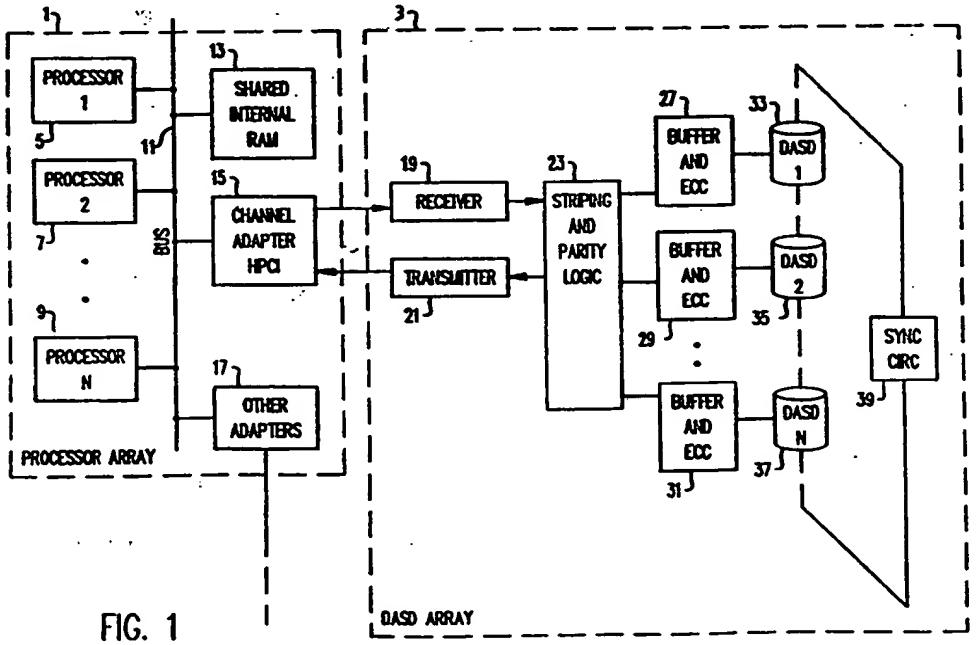

FIG. 1

EP 0 466 339 A2

Field of the Invention

This invention relates to a method and means for passing messages between processors having order of magnitude speed differences to avoid the rate of message exchange being dominated by the lower speed. Such message passing occurs between concurrently executing CPU's and an external storage sub-system (e.g. synchronous direct access storage device (DASD) array).

Background of the Invention

Contemporary high speed processing or supercomputing conjures up the prospect of 1000 million instructions per second (MIPS) of coordinate computing across multiple processors aperiodically referencing substantially slower specialized processors such as direct access storage device (DASD) array controllers. Synchronization among processors still requires a combination of locks and messages. Locks serve to bind resources to tasks while messages and their processing operate as synchronizing events. In contemporary systems, task oriented messages are enqueued against resources. Also, the queued access is governed by a global lock. Thus, operations are paced by the slowest processor obtaining locked access to the queue.

A single central processing unit (CPU) or processor typically includes a local operating system (OS), RAM implemented internal memory, local instruction and data caches operatively formed from the internal memory, an external store, and lock, cache, and storage resource managers. However, high speed or supercomputing involves applications executing over several processors. The applications initiate tasks in the form of OS instructions (READ/WRITE). These tasks are queued against the resources which process them. In this case, these are the general and special purpose processors of the high speed system. The tasks are relatively synchronized (ordered) with respect to each other by their position placement as messages in the queue in a commonly accessible portion of processor shared internal memory.

The messages (tasks) are expressed as encapsulated operations defined over a range of addresses. Where the messages relate to accessing external storage, they are enqueued by processors in the shared memory and await dequeuing and execution by the storage sub-system. Concurrently, messages indicative of altered or completed storage access tasks are also enqueued by the storage sub-system in the shared memory and await dequeuing and execution by the processors. By locking the queue, the slowest processor such as the external store (array controller) can pace the entire operation.

As mentioned above, synchronization is achieved among processors and tasks using queued access messages usually controlled by some form of locking. However, where there is a raw disparity in capacities, then much may go to waste while the faster processor awaits access to a queue currently bound (locked) to a substantially slower processor.

Even where the processor engages external DASD storage without delay, there may be a gross mismatch of data rates. Illustratively, concurrent processors executing 100 MIPS and a 100 MByte/sec data transfer rate might have to communicate with a gigabyte DASD having a 1 to 3 MByte transfer rate and a 10 millisecond access time.

Patterson et al, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", ACM SIGMOD Conference, Chicago Illinois, June 1-3, 1988 discusses the general solution with respect to data rate matching in the form of accessing N synchronized DASDs in parallel. Synchronous behavior requires N DASDs to rotate at the same rpm, have the same angular offset, and be accessed in an identical manner at the same time.

As an alternative to communicating data in parallel via Patterson's synchronized DASD's, data rate mismatch has been managed by interlocks or lockable buffers. Buffer size and cost have remained as obstacles.

Beausoleil et al, USP 3,336,582, "Interlocked Communication System", issued August 15, 1967, shows an interlock over which a low speed processor paces the transfer from a high speed processor. That is, a low speed processor such as a storage control unit (IBM 3880) strobes a CPU/channel (S/370) over a demand response interface indicating that it is available to process the next information unit.

Cage, USP 4,454,595, "Buffer For Use With A Fixed Disk Controller", issued 6/12/1984, discloses a multi-ported random access memory managed by address register manipulation as an asynchronous partitioned circular buffer. Data is read from or written into consecutive RAM addresses on a partition-at-a-time basis in wrap-around (circular) order. A partition/block consists of a fixed number of consecutive RAM addresses sized to hold a track sector of data, the RAM buffer having a capacity of at least two such partitions.

In Cage's buffer, the speed of the movement of fixed blocks of data is matched between the main memory (DMA) of a word processor and an attached DASD. A DASD write or read command results in data being moved either from the DMA or a DASD track sector into a first RAM partition. Because RAM operations are asynchronous, a demand transfer from a second RAM partition can be

overlapped with the first movement as an atomic part of the command (DASD read or write) being executed.

Knuth, "The Art of Computer Programming", Second Edition, copyright Addison-Wesley Pub. Co. 1968, 1973, Vol.1 Fundamental Algorithms, pages 234-239, 531-534, describes a "deque" as a linear list in which all insertions and deletions are made at the ends of the list. He further defines an "input restricted deque" as a linear list in which items may be inserted at one end and removed from the other end (Sec.2.2.1 Exercise 1).

Disclosure of the Invention

The invention provides a method of passing task messages in a data processing system including a plurality of high-speed processors, an external storage subsystem, and shared internal memory, said method comprising the steps of: defining a first linked list in the shared internal memory with separately lockable first and second ends; and operating said list as a queue of task messages passing between the high-speed processors and the storage subsystem, lockably enqueuing task messages at the first end of the queue and independently lockably dequeuing task messages from the second end of the queue.

Preferably the method further comprises the steps of defining a second linked list in the shared internal memory with separately lockable first and second ends; and operating said second list as a queue of task messages passing between the high-speed processors and the storage subsystem, with task messages being lockably enqueued by the external storage subsystem at the first end of the queue, and independently task messages being lockably dequeued by the high speed processors from the second end of the queue. Correspondingly, on the first list task messages are enqueued at said first end of the queue by the high-speed processors, and dequeued from the second end of the queue by the external storage subsystem.

In a preferred embodiment, reference points and operator sets are defined over said first and second lists, one reference point being associated with the second end of each list, and each operator set including top of list (TOQ) and bottom of list (BOQ) pointers, and top of list (TOQL) and bottom of list (BOQL) lockwords. A first lock is obtained by one of the processors on said second end of the first list upon said one processor matching the BOQL, whereupon task messages are embedded between the reference point and the last message in the list, and said first lock is released. A second lock can be obtained by the subsystem on said first end of the first list upon said subsystem matching the TOQL, allowing one or more task messages to be removed from said first list before said second lock is released. Only the holder of said second lock (TOQL) may alter any pointer to the non-reference end of the list or any of the task messages in the list, and only the holder of the first lock (BOQL) may alter any pointer designating the reference point end of the list.

It is also preferred that the processors originate task messages in the form of control blocks (DCB's) for the subsystem, each DCB specifying one or more access operations to be executed by said subsystem, said DCB's being enqueued on the first list by the originating processors, and said subsystem dequeuing and processing each DCB from said first list according to either an external discipline (LIFO, FILO, FIFO) or according to a priority reordering of said first list or portion thereof. Each DCB assumes either a waiting, active, or completed status, the waiting DCB's constituting the first list, active DCB's being currently processed, and completed DCB's constituting the second list. The subsystem then processes task messages dequeued from said first list, updates each task message, and enqueues the updated task messages onto the second list, said originating processors dequeuing and processing each updated task message from said second list according to either an external discipline (LIFO, FILO, FIFO) or to a priority reordering of said second list or portion thereof.

The invention also provides a data processing system including a plurality of high-speed processors, an external storage subsystem, and shared internal memory, said system further including: means for defining first and second linked lists in the shared memory each with separately lockable first and second ends; means for operating said first and second lists as wait and completion queues of task messages respectively, lockably enqueuing task messages at the first end of a queue and independently lockably dequeuing task messages from the second end of the queue; means responsive to task messages originating from processors for writing the task messages into the shared memory, linking said task messages into the wait queue, and signalling the storage subsystem accordingly; means at the storage subsystem aperiodically responsive to said signalling for dequeuing the task messages from the wait queue, actively processing the dequeued task messages, enqueuing the processed task messages onto the completion queue in said shared memory, and signalling the processors accordingly; and means aperiodically responsive to the subsystem signals for dequeuing the processed task messages from the completion queue.

The above method and means permit processors of disparate speed to have overlapped access

to either add or remove messages in queued access (position order processing) relation to each other. Thus processors such as a CPU and an external storage subsystem placing a message on a queue no longer have to wait on a queue lock set by another processor or subsystem dequeuing a message. A double ended linked list or queue of messages is used, preferably with an isolation/reference point (NULL/BLANK). An enqueuing end of the list (BOQ/BOQL) is lockable and accessible independently from the dequeing end of the list (TOQ/TOQL). Typically the method and means utilize simple meta-processor lock primitives (in this regard, "meta-processor" denotes available to all processors). The locking primitive may be of the multi-processor lock synchronizing atomic type such as TEST AND SET. The processor obtains a lock, when available, before inserting messages between the first end and the last message on the first queue, and then releases the lock. Likewise the subsystem obtains a lock, when available, before inserting messages between the first end and the last message on the second queue. The availability of a lock can be tested by comparison matching BOQL with the result from executing the lock primitive.

Thus, in other words, task oriented messages can be passed between a plurality of high speed processors and a relatively low-speed external storage subsystem communicatively coupled over a shared memory, using oppositely poled queues. First and second dense linked linear lists are defined in the shared memory, each list having independently lockable first and second ends. A first lock is obtained by a processor on the first end of the first list when available, the processor inserting messages between the first end and the last message so linked, and releasing said first lock. The subsystem obtains another lock on the second end of the first list when available, removing messages anywhere on the list, and releasing said other lock. On the first and second ends of the second list the roles of the storage sub-ystem and processors are reversed.

Brief Description of the Drawings

Figure 1 shows a CPU/ DASD array data flow emphasizing shared internal memory, high performance channel interface (HPCI), array controller, and DASD's.

Figures 2A-C respectively depict a wait queue, an active list, and a completion queue of DASD control blocks (DCB's) selectably lockable at either end thereof.

Figure 3 illustrates the enqueue operation on a DCB wait and completion queue.

Figure 4A-B sets forth the dequeue operations on a respective first and second example of a DCB wait and completion queue.

Detailed Description

The preferred embodiment of this invention uses a high speed multi-processor host interacting with a slower external store. The external store is illustratively expressed as a synchronous array of N DASD's and an array controller. To enhance appreciation of this form of external storage, a brief description of data organization on the array (striping) and the use of information redundancy (parity blocks, ECC) is provided.

Patterson's type 3 DASD array synchronously reads and writes to N DASDs in column major order. However, N-1 of the DASD's contain data and one DASD contains a parity ranging over the other data DASDs. That is, one check DASD is provided for the group. The contents of the failed DASD can be reconstructed in the manner of Ouchi, US Pat 4,092,732, "System for Recovering Data Stored in a Failed Memory Unit", issued May 30, 1978, which discloses the spreading of data blocks from the same logical file across a string of N- 1 DASD's and recording a parity block on the Nth DASD. The parity block is an XORing of the parity contents of the N-1 other blocks. Contents from any single inaccessible DASD can be recovered by XORing the parity blocks with the blocks stored on the N-2 remaining accessible DASD's. A similar result can be achieved if the parity blocks are not available.

Typically in block oriented data, a parity suffix or equivalent (Hamming, CRC) is appended to each data block. Thus, each parity suffix can be invoked to detect/correct intra-block error. As described in Ouchi, when one or more of the blocks of an N-1 sequence are unavailable, a parity block, which a priori spans the N-1 block sequence, is used in conjunction with the remaining blocks to rebuild the unavailable data block. Efficient codes per se (Hamming, Cyclic Redundancy Check, Reed-Solomon) are elsewhere treated in the literature.

Referring now to Figure 1, there is shown a system including a an array of processors 1 coupling as external storage a synchronous DASD array 3. The processors 5, 7, 9 are of the high performance variety such that when operated concurrently they have processing speeds in the order of 100+ MIPS. A slower special purpose processor in the form of DASD array 3 is coupled to array 1 by way of adapter 15. Other information processing sources or sinks such as local area networks, printers, or displays would likewise be coupled over counterpart adapters 17.

The fast and slow processors communicate

within array 1 over a very high speed bus 11 using a portion of shared internal RAM 13 as a specialized message repository. The processors constitute a distributed peer coupled system with no centralized operating system or process control. All resources such as internal memory 13, interrupt facility, and global registers (not shown) are available to any processor including the external storage or DASD array subsystem (DAS) 3.

DAS 3 preferably comprises a RAID3 type DASD array and an associated array controller as described for example in the Patterson reference and in the co-pending Brady et al. application, EP 91304503.5, "METHOD AND MEANS FOR ACCESSING DASD ARRAYS WITH TUNED DATA TRANSFER RATE AND CONCURRENCY", (Applicant's ref. no. SA9-90-028). Even though DAS operates in a peer-coupled relationship with the other processors, it nevertheless queues task oriented messages against other processors or resources. Any change in the queue or other action requests such as halt current operations are indicated by way of special purpose signals (tap signals).

DAS receives tasks as messages and communicates the results of task processing via control blocks arranged in the form of a dedicated queue. This queue, as exemplified in Figure 2, remains in host 1 internal memory 13. The system includes a facility to reorder the queue according to changing system priorities.

Referring again to Figure 1, there is shown an array controller (elements 19-31) coupling the host bus 11 by way of channel adapter 15. This path provides access to all system resources including memory 13, global registers, and tap signals on behalf of DAS 3. Adapter 15 preferably attaches DAS over a pair of simplex megabyte rate receive and transmit interfaces 19, 21. The interfaces, known also as High Performance Parallel Interfaces (HIPPI), are described in the ANSI Draft Standard of 8/29/1989, X3T9/88-127, Revision 6.8. This facilitates receipt of the so-called tap signals from the host and access to the host as initiated by DAS.

Host or system functions available for DAS use include READ/WRITE internal memory 13, atomic TEST AND SET for lock operations in internal memory 13, atomic operations on global registers, receipt and interpretation of tap signals from other processors or system elements, and the generation of tap signals to the system or host.

DAS 3 operates one addressable array of DASD's 33, 35, 37 synchronized via sync circuit 39 to each rotate at the same rpm, have the same angular offset, and be accessed in an identical manner at the same time. This solution maximizes data transfer rate. This permits high speed sequential or skip sequential DASD data transfer.

Data movement in the Host/DAS direction starts from internal memory 13 over bus 11 through adapter 15, receiver 19, ending in striping and parity logic 23. Logic 23 includes the necessary digital and timing circuits to calculate a parity block by XORing the N-1 data blocks and transferring counterpart blocks to ones of the buffer and ECC circuits 27, 29, and 31. Each block is also protected by appending an ECC byte thereto for error detection and correction on an intra-block basis. Access to the N DASD's is made synchronously in the conventional manner. The obverse obtains when data movement proceeds in the DAS/Host direction.

The task or request is defined in a control block termed a DASD Control Block or DCB. A processor at the host builds a DAS DCB in internal memory 13. It then links the DCB to a WAIT queue for DAS and signals the enqueuing operation. Subsequently, DAS moves the DCB onto the ACTIVE LIST, performs DCB requested functions, updates the DCB and places it on the COMPLETION queue. After detecting the I/O completion, the host dequeues the updated DCB from the COMPLETION queue and ascertains the outcome of the DCB specified operation.

Restated, a DAS I/O request is responsive to any processor 5-9 invoking a READ/WRITE from its OS. A DCB is built and placed on a WAIT queue located in internal memory 13. A tap signal is then sent to DAS. In turn, the DAS inspects the queue for the next queued DCB using any one of a number of work management algorithms (FIFO, FILO, LIFO etc.). DAS does not need to respond to the tap signal as a priority event.

Referring now to Figures 2A-C, the system is arranged such that the DAS "task WAIT queues" are priority ordered. A request may be relocated in the queue relative to other tasks or DCB's at any time prior to its being made active by the DAS. In this invention, such reordering is a special function of a DEQUE operation to be explained subsequently. Note, that the DCB's remain at the same internal memory 13 real address irrespective of changes in queue linking priority or active/waiting status.

One of the attributes of peer-coupled processors is that the DAS is responsive to receipt either of a tap signal or its work management algorithm by finding the next queued DCB via reading an anchor pointer in internal memory 13 established during initialization and changing the first waiting DCB status from "WAITing" to "active. This is accomplished by moving the DCB show in Figure 2A to the ACTIVE LIST per Figure 2B. This is brought about by the DAS executing a sequence of internal memory access operations. Once the DCB is "active", the DAS processes the DCB according

a function code contained in said DCB. Relatedly, address information in the DCB define the extent of data to be transferred. In this regard, the data transfer is effectuated by DAS initiated operations through the channel adapter 15.

It should be appreciated that each DCB can assume one of three states, namely, "active", "waiting", or "completed".

When a data transfer operation is completed, the DAS writes completion status in the DCB, changes the DCB state as per Figure 2B from "active" to "completed", and enqueues the DCB on the associated completion queue per Figure 2C. Also, the DAS may execute a number of DCB specified completion notification primitives. Significantly, the transfer of data to and from internal memory 13 and the DAS is under DAS control.

Referring still to Figures 2A-C, there are shown several queues of DCB's. Each queue element, generically called QEL, is a contiguous set of words in a memory which both processor classes can fetch from and store to in shared memory 13. A QEL includes message words as well as a link pointer word required for queueing. Although each QEL could be a different length, advantageously a simple system might use uniform sized QELs of, perhaps, 32 words each. Typically, each shared memory word itself consists of at least enough bit positions to contain the address of any other word. For example each word of shared memory could be 64 bits long.

A pointer to a QEL is the address of one of the words in the QEL (as used in this specification and Figures 2A-C, the term "pointer" refers either to an address, or to a place where such an address is found). Context determines which is meant. The word pointed to, the link pointer word, normally contains the address of another QEL, but can contain a predetermined null value such as all bits off, or all on.

As illustrated Figures 2A-C, a Blank-QEL is a QEL whose link pointer word is null. Also, a queue may be formed from either a single Blank-QEL, or a set of QELs linked one to another by pointers leading from a top QEL to a bottom QEL. The latter is always a Blank-QEL when the bottom pointer is unlocked.

Queues are locked by using a multiprocessor synchronizing lock operation available to all participating multiprocessor classes. An example is the classic Test and Set operation which stores a lock constant in a word of shared memory. If the preceding fetch, which is part of the same atomic operation, returns a value other than this lock constant, then the Test and Set operation has successfully locked the lock word.

A queue header consists of four words in the shared memory as shown following:

TOQ	Top of Queue Pointer: Shared memory address of link pointer word of the top QEL in the queue.
BOQ	Bottom of Queue Pointer: Shared memory address of link pointer word of the bottom QEL in the queue.
TOQL	TOQ Lock Word. For example a Test and Set lock word. Only the holder of the TOQ lock may alter the TOQ pointer or any of the contents of QELs linked above the QEL pointed to by BOQ. (Note that the BOQ pointer may change while TOQ is locked; see BOQL)
BOQL	BOQ Lock Word. Only the holder of the BOQ lock may alter the contents of the QEL at the bottom of the queue, possibly making it non-blank. Only the BOQL processor/holder may change the BOQ pointer value, and then only if the new value is the address of a valid Blank-QEL.

Changing the BOQ pointer moves the isolation point between the enqueuing and dequeuing processes. Once the BOQ is changed, the BOQ lock holder's authority to update is reduced to the New-Blank-QEL and to the BOQ pointer. A TOQ lock holder is always free to update any QELs down to, but not including, the QEL pointed to by the BOQ, whether the BOQ is locked or not.

Applying the above definitions to Figures 2A-C, there are shown constructs indicative of the three states into which each DCB must be resolved (wait, active, completed). DCB's classified as waiting or completed are enqueued while an active DCB is being processed. In each queue, each DCB has a pointer to the next DCB in the chain. The BOQ points to a null or blank DCB operative as the queue bottom while TOQ points to the most recent queue addition.

Referring now to Figure 3, there is depicted an enqueue operation as the loading of a message into the Blank-QEL at the bottom of a queue, and the appending to the bottom of the queue of an additional QEL as the New-Blank-QEL. The enqueue operation includes the steps of:

1. locking BOQ, conditionally waiting for the queue to become available; 2. writing a null into link pointer of the additional QEL, "New-Blank"; 3. writing the memory address of New-Blank into Linkage Pointer of Old-Blank; 4. loading a message into the former Blank-QEL, "Old- Blank"; 5. putting memory address of New-Blank into the BOQ pointer word; and 6. unlocking BOQ.

The original queue has not been altered except for the contents of the original Blank-QEL. The original Blank-QEL was initially at the bottom of the queue. Only the processor/holder of the TOQ lock

may alter the queue's non-blank QELs.

The enqueuing processor can add more than one QEL. In preparation for enqueuing, the additional QELs are to be linked one to another from Top-Addnl-QEL to Bot-Addnl-QEL. The steps are listed below in an order which continuously maintains a well formed queue. Note, a "well formed queue" refers to a TOQ-BOQ pair and its associated queue conforming to the above definitions.

1. locking BOQ, perhaps waiting for it to become available;
2. putting null into link pointer of Bot-Addnl-QEL, making it New-Blank;
3. loading messages into Old-Blank, and all Addnl-QELs except New-Blank;
4. putting memory address of Top-Addnl into link pointer of Old Blank;
5. putting address of New-Blank into BOQ; and
6. unlocking BOQ.

Referring now to Figures 4A-B, there are depicted two examples using a DEQUE operation in which a single QEL is removed from the top of a queue. A processor copies into the TOQ the contents of the link pointer from the first QEL in the queue. To remove a continuously linked sequence of QELs, a processor changes the TOQ pointer, or a single QEL link pointer, so as to point around the QELs being removed. The dequeue steps are listed below in an order which continuously maintains a well formed queue.

1. if TOQ = BOQ, exiting without locking as queue is empty of non-blanks;
2. locking TOQ, conditionally after waiting for it to become available;
3. reading BOQ, any QEL at that address or linked beyond it is ineligible;
4. removing one or more eligible QELs from anywhere on the queue; and
5. unlocking TOQ.

The QEL pointed to by BOQ, when BOQ is read, is treated as ineligible. While a BOQ lock holder may make this a valid QEL and may change BOQ, the QEL will remain ineligible for this execution of enqueue or reorder.

Reordering is a two step process in which QEL(s) are first dequeued from, and then reinserted into, the eligible QEL chain. The processor doing the reordering holds the TOQ lock for the duration of both steps. So to move a single QEL, called "Moving-QEL", to the top of the queue, a processors dequeues the QEL, as defined previously, and then reinserts it by changing Moving-QEL's link pointer to the value in TOQ, and putting the address of Moving-QEL into TOQ.

In a similar manner a processor can dequeue any number of eligible QELs, link them together in one or more fragmentary chains, and then reinsert the fragments back into the queue. Reinsertion can be done maintaining a well formed queue by:

1. setting link pointer of fragment's final QEL to address of QEL that will follow fragment in queue; and
2. setting TOQ pointer, or link pointer of QEL that is to precede fragment, to the address of first QEL in fragment.

For any operation to continuously maintain a well formed queue, each single word storage update of a pointer in shared memory must complete fully. If a process then follows the steps of an operation in the order given, the queue will remain well formed. In the event of most hardware failures, or of a premature process termination, some in-transit QELs may not be on the queue, but the queue itself will remain well formed. Other processes will be able to continue working with such a queue.

In the initial example the allocation of shared memory can be a processor only function. This can be achieved by having the controllers reuse each work queue element as the QEL in which to report completion status.

Claims

1. A method of passing task messages in a data processing system including a plurality of high-speed processors (5, 7), an external storage subsystem (3), and shared internal memory (13), said method comprising the steps of:

   defining a first linked list in the shared internal memory with separately lockable first and second ends; and

   operating said list as a queue of task messages passing between the high-speed processors and the storage subsytem, lockably enqueuing task messages at the first end of the queue and independently lockably dequeuing task messages from the second end of the queue.

2. A method as claimed in claim 1, in which the task messages are enqueued at said first end of the queue by the high-speed processors, and dequeued from the second end of the queue by the external storage subsystem.

3. A method as claimed in claim 2, further comprising the steps of:

   defining a second linked list in the shared internal memory with separately lockable first and second ends; and

   operating said second list as a queue of

task messages passing between the high-speed processors and the storage subsystem, task messages being lockably enqueued by the external storage subsystem at the first end of the queue, and independently task messages being lockably dequeued by the high speed processors from the second end of the queue.

4. A method as claimed in claim 3, further comprising the steps of:

defining reference points and operator sets over said first and second lists, one reference point being associated with the second end of each list, and each operator set including top of list (TOQ) and bottom of list (BOQ) pointers, and top of list (TOQL) and bottom of list (BOQL) lockwords;

obtaining a first lock by one of the processors on said second end of the first list upon said one processor matching the BOQL, embedding task messages between the reference point and the last message in the list, and releasing said first lock;

obtaining a second lock by the subsystem on said first end of the first list upon said subsystem matching the TOQL, removing one or more task messages from said first list, and releasing said second lock.

5. A method as claimed in claim 4, wherein only the holder of said second lock (TOQL) may alter any pointer to the non-reference point end of the list or any of the task messages in the list, and further wherein only the holder of the first lock (BOQL) may alter any pointer designating the reference point end of the list.

6. A method as claimed in any preceding claim, wherein said method further comprises the step of:

reordering the list of task messages by obtaining a lock on the second end of the first list when available, removing task messages from anywhere on the list, linking at least some of the dequeued task messages together in one or more fragmentary chains, and then reinserting the fragmentary chains back into the queue, and releasing the lock.

7. A method as claimed in any of claims 2 to 6, wherein the processors originate task messages in the form of control blocks (DCB's) for the subsystem, each DCB specifying one or more access operations to be executed by said subsystem, said DCB's being enqueued on the first list by the originating processors, and said subsystem dequeuing and processing each DCB from said first list according to either an external discipline (LIFO, FILO, FIFO) or according to a priority reordering of said first list or portion thereof.

8. A method as claimed in claim 7, wherein each DCB assumes either a waiting, active, or completed status, the waiting DCB's constituting the first list, active DCB's being currently processed, and completed DCB's constituting the second list.

9. A method as claimed in claim 7 or 8, wherein the subsystem processes task messages dequeued from said first list, updates each task message, and enqueues the updated task messages onto the second list, said originating processors dequeuing and processing each updated task message from said second list according to either an external discipline (LIFO, FILO, FIFO) or to a priority reordering of said second list or portion thereof.

10. A data processing system including a plurality of high-speed processors (5, 7), an external storage subsystem (3), and shared internal memory (13), said system further including:

means for defining first and second linked lists in the shared memory each with separately lockable first and second ends;

means for operating said first and second lists as wait and completion queues of task messages respectively, lockably enqueuing task messages at the first end of a queue and independently lockably dequeuing task messages from the second end of the queue;

means responsive to task messages originating from processors for writing the task messages into the shared memory, linking said task messages into the wait queue, and signalling the storage subsystem accordingly;

means at the storage subsystem aperiodically responsive to said signalling for dequeuing the task messages from the wait queue, actively processing the dequeued task messages, enqueuing the processed task messages onto the completion queue in said shared memory, and signalling the processors accordingly; and

means aperiodically responsive to the subsystem signals for dequeuing the processed task messages from the completion queue.

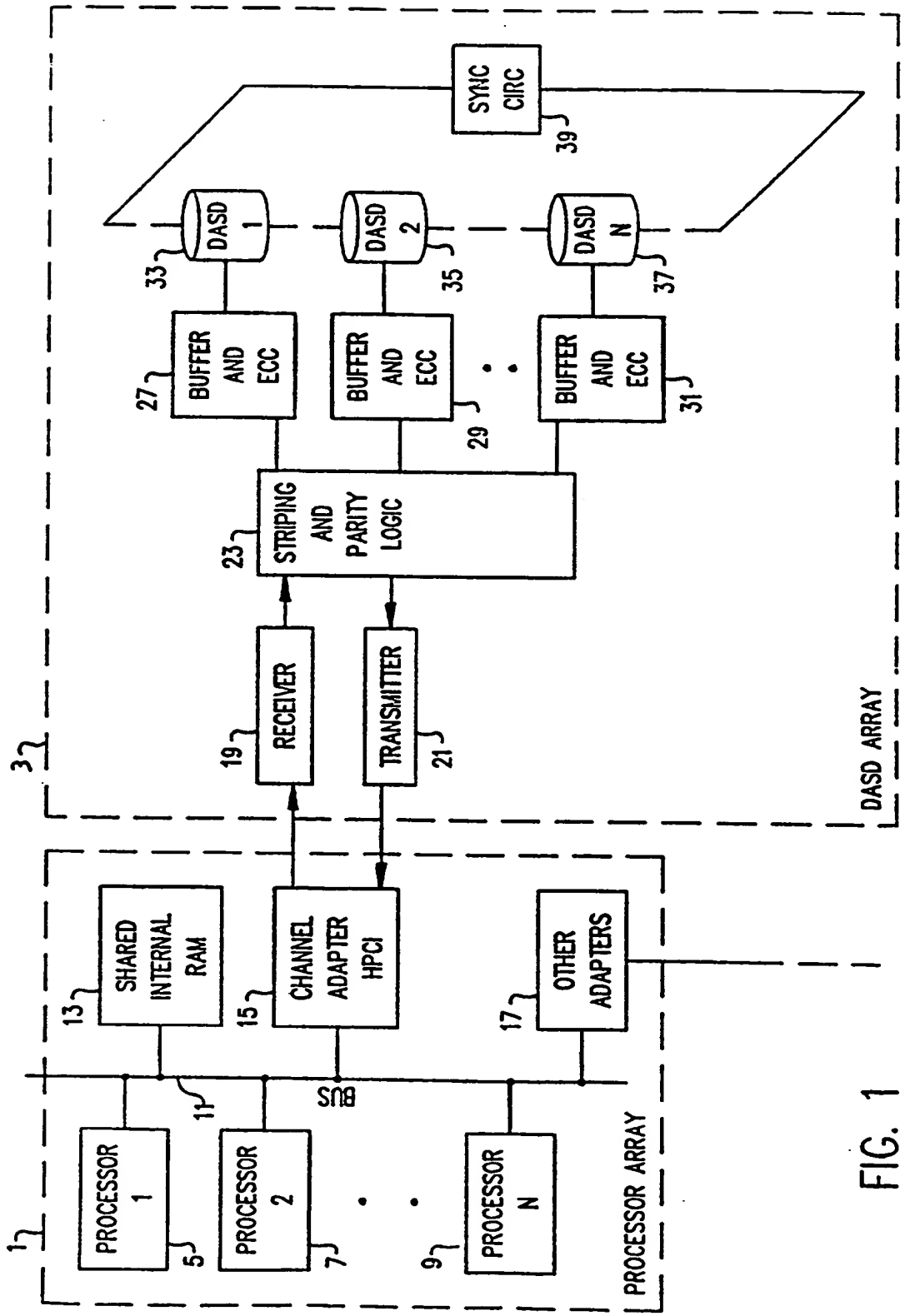
SYNC CIRC
39
DASD 1
DASD 2
DASD N
33
35
37
BUFFER AND ECC
27
BUFFER AND ECC
29
BUFFER AND ECC
31
STRIPING AND PARITY LOGIC
23
RECEIVER
19
TRANSMITTER
21
3
DASD ARRAY
SHARED INTERNAL RAM
13
CHANNEL ADAPTER HPCI
15
OTHER ADAPTERS
17
BUS
11
PROCESSOR 1
5
PROCESSOR 2
7
PROCESSOR N
9
1
PROCESSOR ARRAY

FIG. 1

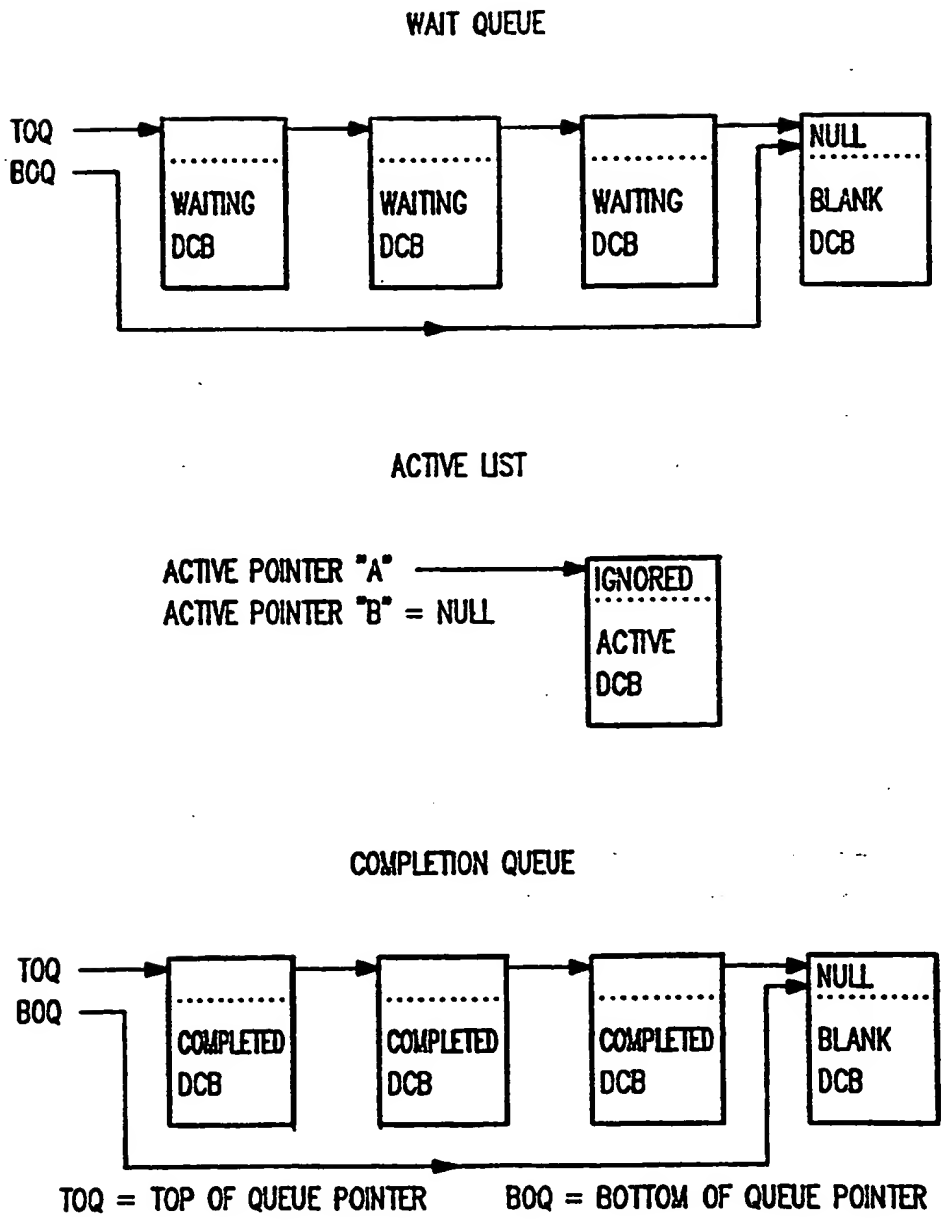

FIG. 2

ENQUE

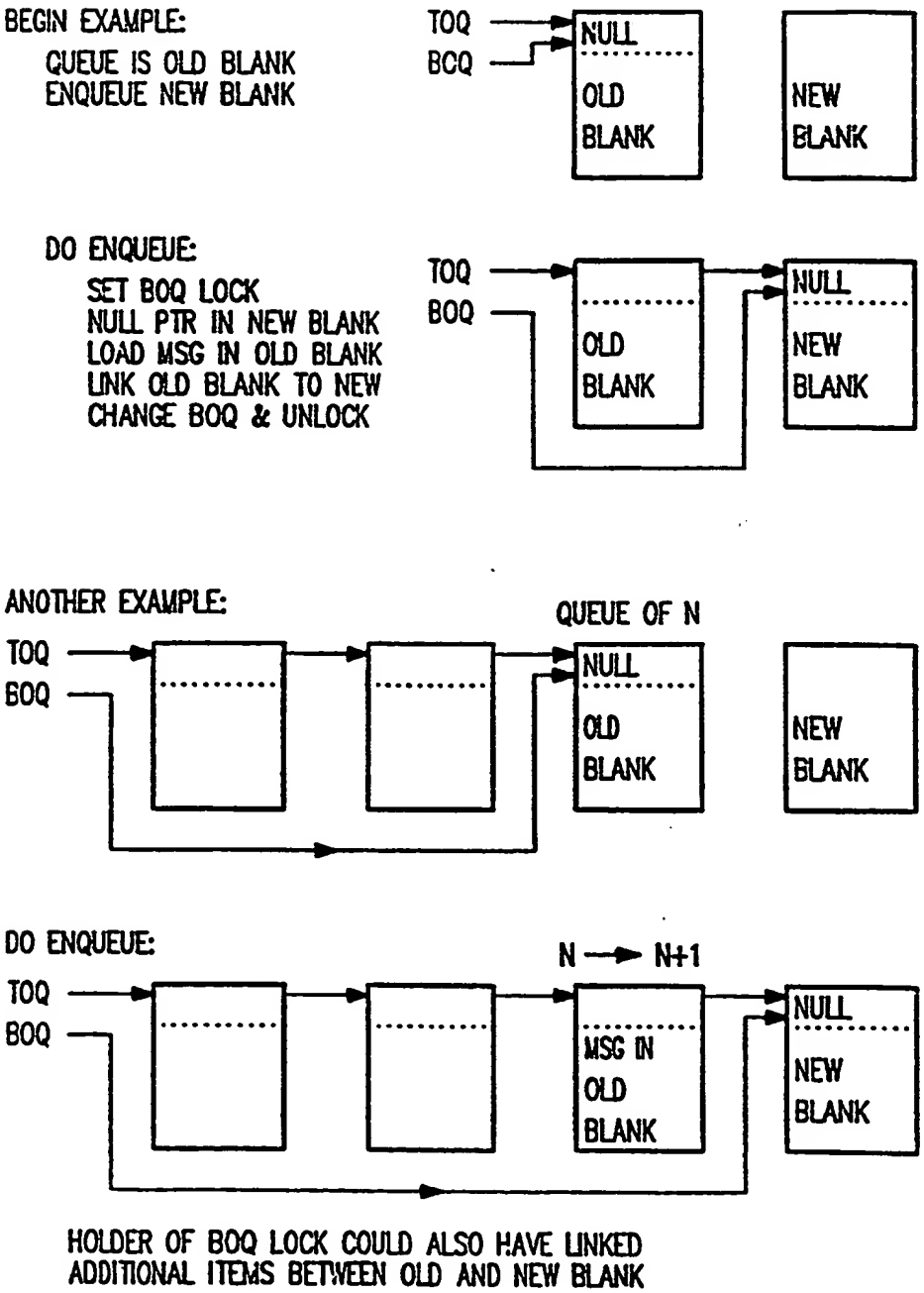
BEGIN EXAMPLE:
QUEUE IS OLD BLANK
ENQUEUE NEW BLANK
TOQ
BCQ
NULL
OLD
BLANK
NEW
BLANK
DO ENQUEUE:
SET BOQ LOCK
NULL PTR IN NEW BLANK
LOAD MSG IN OLD BLANK
LINK OLD BLANK TO NEW
CHANGE BOQ & UNLOCK
TOQ
BOQ
NULL
OLD
BLANK
NEW
BLANK
ANOTHER EXAMPLE:
QUEUE OF N
TOQ
BOQ
NULL
OLD
BLANK
NEW
BLANK
DO ENQUEUE:
N → N+1
TOQ
BOQ
NULL
MSG IN
OLD
BLANK
NEW
BLANK
HOLDER OF BOQ LOCK COULD ALSO HAVE LINKED
ADDITIONAL ITEMS BETWEEN OLD AND NEW BLANK

FIG. 3

DEQUEUE

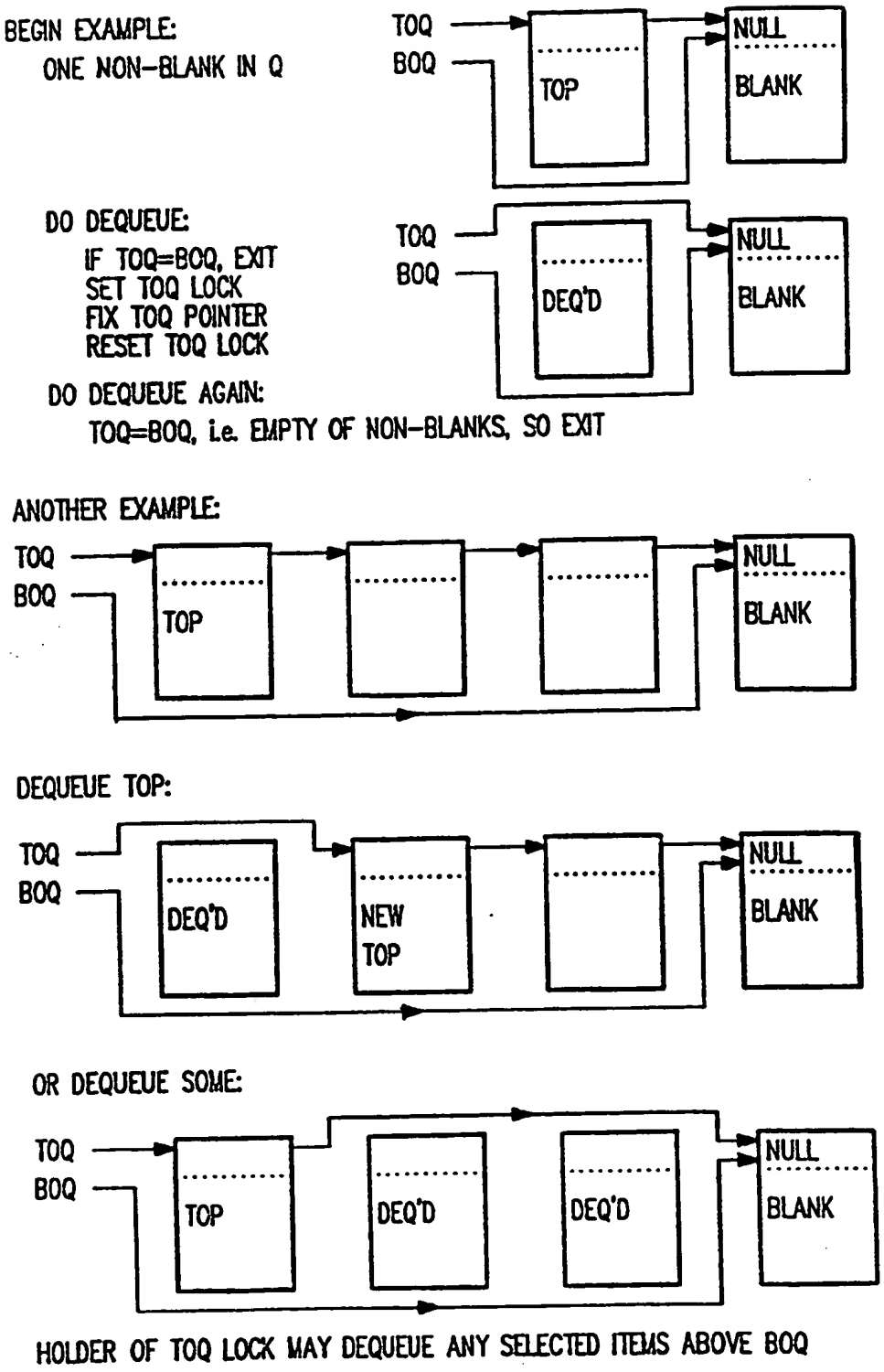

FIG. 4

(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) Publication number: **0 466 339 A3**

(12) EUROPEAN PATENT APPLICATION

(21) Application number: **91305396.3**

(22) Date of filing: **14.06.91**

(51) Int. Cl.5: **G06F 9/46, G06F 5/06, G06F 7/00, G06F 3/06**

(30) Priority: **13.07.90 US 553203**

(43) Date of publication of application: **15.01.92 Bulletin 92/03**

(84) Designated Contracting States: **DE FR GB**

(88) Date of deferred publication of the search report: **11.08.93 Bulletin 93/32**

(71) Applicant: **International Business Machines Corporation**
Old Orchard Road
Armonk, N.Y. 10504(US)

(72) Inventor: **Disbrow, John Randolph**
16500 So. Kennedy Road
Los Gatos, CA 95032(US)

(74) Representative: **Moss, Robert Douglas**
IBM United Kingdom Limited Intellectual Property Department Hursley Park
Winchester Hampshire SO21 2JN (GB)

(54) A method of passing task messages in a data processing system.

(57) Processors 5, 7 communicatively attached to a storage sub-system 3 place task messages for the subsystem on a queue. The processors do not have to wait on a queue lock set by another processor or sub-system whilst dequeuing a message. This is achieved by use of a double ended linked list or queue of messages having an isolation/reference point wherein an enqueuing end of the list is lockable and accessible independently from the dequeuing end of the list. The locking primitive may be of the multi-processor lock synchronizing atomic type such as TEST AND SET.

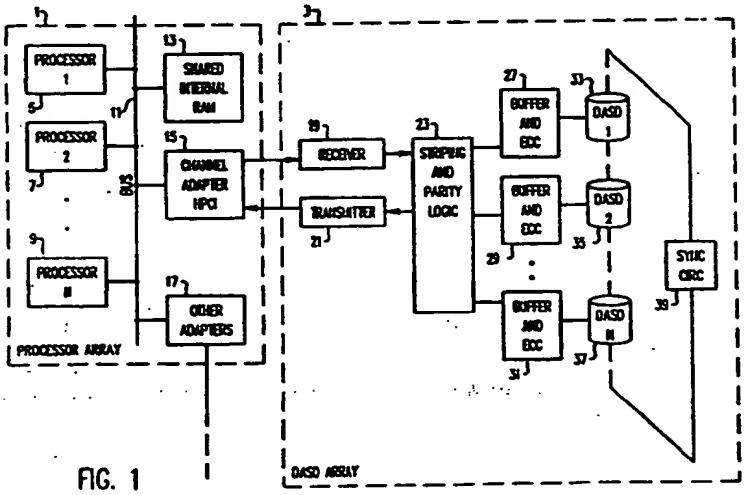

FIG. 1

EP 0 466 339 A3

European Patent Office

EUROPEAN SEARCH REPORT

Application Number

EP 91 30 5396

DOCUMENTS CONSIDERED TO BE RELEVANT

Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int. Cl.5)
X	IBM TECHNICAL DISCLOSURE BULLETIN. vol. 24, no. 1A, June 1981, NEW YORK US pages 363 - 364 B.C. GOLDSTEIN ET AL 'Atomic head and tail pointer manipulation on double-threaded queues' * the whole document *	1-3	G06F9/46 G06F5/06 G06F7/00 G06F3/06
Y		4-7,9	
Y	EP-A-0 205 946 (INTERNATIONAL BUSINESS MACHINES)	4,5,7,9	
A	* column 2, line 6 - line 54; claim 1; figure 1 *	1-3,8	
Y	D.E. KNUTH 'The art of computer programming, Vol. 3, Sorting and Searching' 1973 , ADDISON-WESLEY , READING, MA, US * Par 5.2.5 Sorting by distribution * * page 170, line 12 - page 141, line 31 *	6	

TECHNICAL FIELDS SEARCHED (Int. Cl.5)

G06F

The present search report has been drawn up for all claims

Place of search	Date of completion of the search	Examiner
THE HAGUE	11 JUNE 1993	KINGMA Y.

CATEGORY OF CITED DOCUMENTS

X : particularly relevant if taken alone
Y : particularly relevant if combined with another document of the same category
A : technological background
O : non-written disclosure
P : intermediate document

T : theory or principle underlying the invention
E : earlier patent document, but published on, or after the filing date
D : document cited in the application
L : document cited for other reasons

& : member of the same patent family, corresponding document

EPO FORM 1503 03.82 (P0401)